CORK INSTITUTE OF TECHNOLOGY
INSTITIÚID TEICNEOLAÍOCHTA CHORCAÍ

Mobile Robotics

Project

Albert MarquillasEstruch

Table of contents

Introduction

The aim of this project is to implement a fully working program for a robot, this program will consist of a group of different behaviors, which first will be implemented separately, and then, when they are working alone, fuse them into the same main program. These behaviors will be able to run independently and will do a different and a specific set of operations. When they are implemented inside the same program, it will seem that the robot is very clever doing different actions and changing between behaviors, when it is not.

What will help us to implement this is the subsumption architecture, which later will be explained how it was implemented inside the project, and basically it is a way to implement high-level control of a robot. The basis of this is to spread the actions between different behaviors and each of them requests the control of the robot when some triggering action occurs. Then the arbiter, decides to give them the control or not, depending on their priority, for example: a behavior with a low priority and a behavior with a very high priority want the control at the same time, if this happens, the arbiter will give the control to the high priority one, because not doing so could end with a very big problem than not giving the control to the low priority one. Also, there is always a default behavior which is constantly running, when any behavior requests and gets the control, after finishing its actions, then it returns to the default behavior.

Every part of the architecture will be done with LabView using the LEGO NXT robotics package to control the robot, and it goes without saying that the robot controlled will be a LEGO NXT robot.

Behaviors

As said before, this project uses subsumption architecture, which means that it will have a set of different behaviors, there will be a total of 8 different ones, including the initialization, the termination and the default one. In all of them there is an instruction that is the same, which is displaying the number of the behavior in the NXT screen, this is done in order to provide information to the user to be able to know what is the robot exactly doing, having said so, when describing each of the behaviors, I will not talk about this instruction.

There are some design decisions that affect all the behaviors, these are listed below:

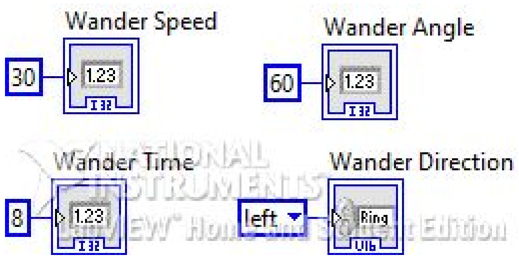
- Each behavior is implemented as an individual program, which can be run in the robot, and it will execute the activities of the behavior.
- For all the behaviors I used a specific ring where every behavior had its own numerical value, so when displaying the behavior number through the screen it will be easier to implement it.
- For the Wander Direction parameter I created a specific ring with “right” and “left”, I decided to use 1 and 0 for their values and then when setting the rotation speed I compared this value, and then multiplied by 1 or -1 the speed. Later, I discovered that setting their values directly to 1 and -1 will ease its use, but as I had most of the behaviors finished I decided to continue as it was, even if it is not the most optimum way to do it.
- For rotating the robot I decided to reuse one of the exercises done in class as I knew exactly how it was working, I used it as an imported VI.

Each behavior has its own different set of actions, and they will be explained for all in the following lines.

Initialization

The initialization does not have any special action to do, it basically initializes the values of Wander Angle, Wander Speed, Wander Direction and Wander Time to some default values that will be used in the different behaviors. It will only be executed once.

The block diagram is the following:

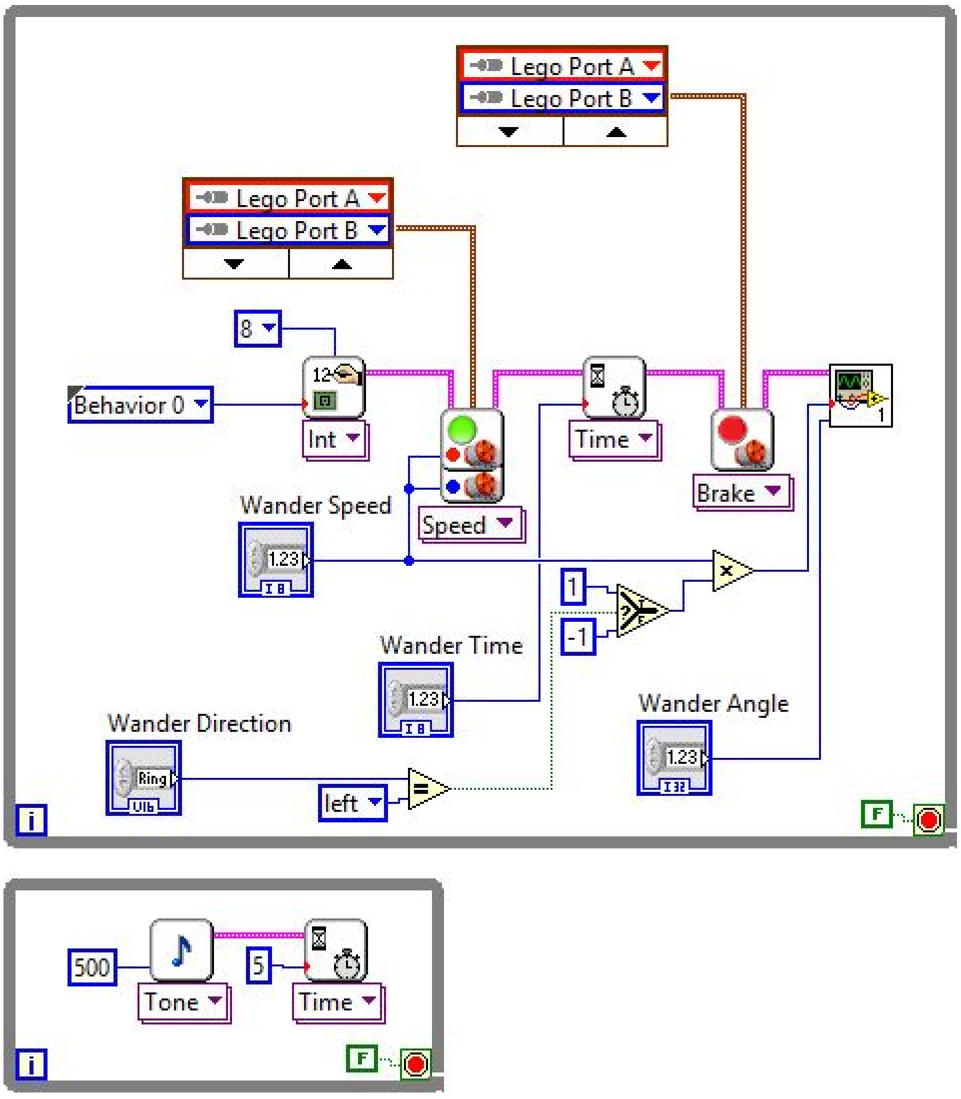


This block worked well, as its complexity is very low, it was tested by running it with the main application, and then check if the indicators in the front panel had the values desired or not.

Behavior 0

This is the default behavior, if any other behavior does not request the control this will be the one making the robot do some actions. The actions are run in a straight line for a “Wander Time” time and with a “Wander Speed” speed, then it rotates “Wander Angle” degrees in the direction specified by “Wander Direction”. Also, while the robot is doing these actions, in parallel is beeping.

The block diagram of this behavior is the following:

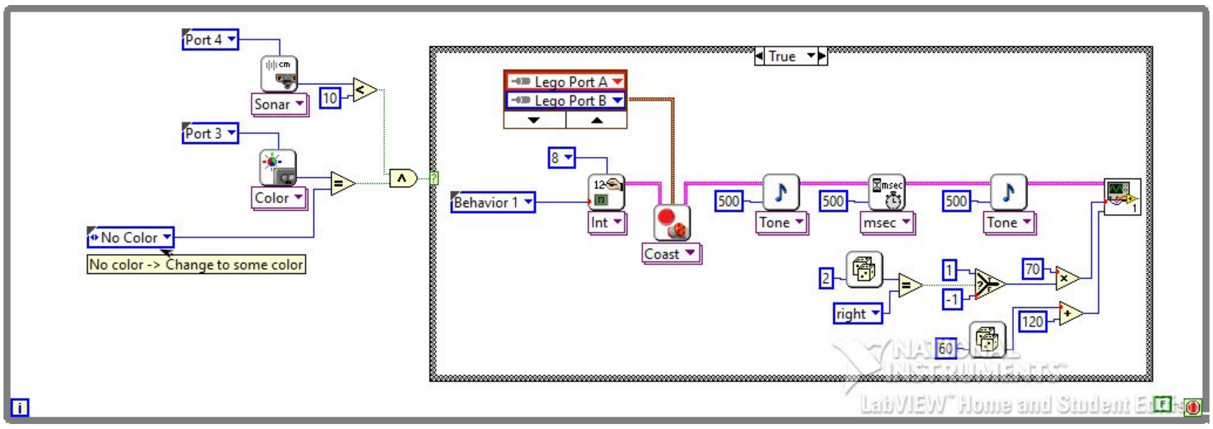


Even this behavior had more complexity than the Initialization one, it did not create any problem. The first test was successful.

Behavior 1

This behavior is entered only when the ultrasonic sensor detects an object in less than 10 cm, and no color is detected, this no color will be the color of the floor, which means that in the real usage this will be changed to a specific color, and also if none of the higher priority behaviors requests the control.

The actions that this behavior will do when it gets the control of the robot are stopping the motors, then playing two tones of 0.5 seconds long with a wait of 0.5 seconds between them, and to finish, it rotates a random angle between 120º and 180º in a random direction.



In this behavior the problems started, as this one had more complexity than the others. The most problematic thing to do was the generation of random values for the rotation. The problem with this was that using the random generator from LabView and not the NXT one, did not work as expected, so I used the NXT random number generator to create a number from 0 to 1 and then compare it and multiply the speed by 1 or -1 depending on the result.

Another problem here was also due to the random number generator, and this time was due to the random value for the angle, as it is only able to generate a value from 0 to the value specified, and we wanted a value from 120º to 180º, I had to generate a number from 0 to 60, and add it to 120, so the random angle value is achieved.

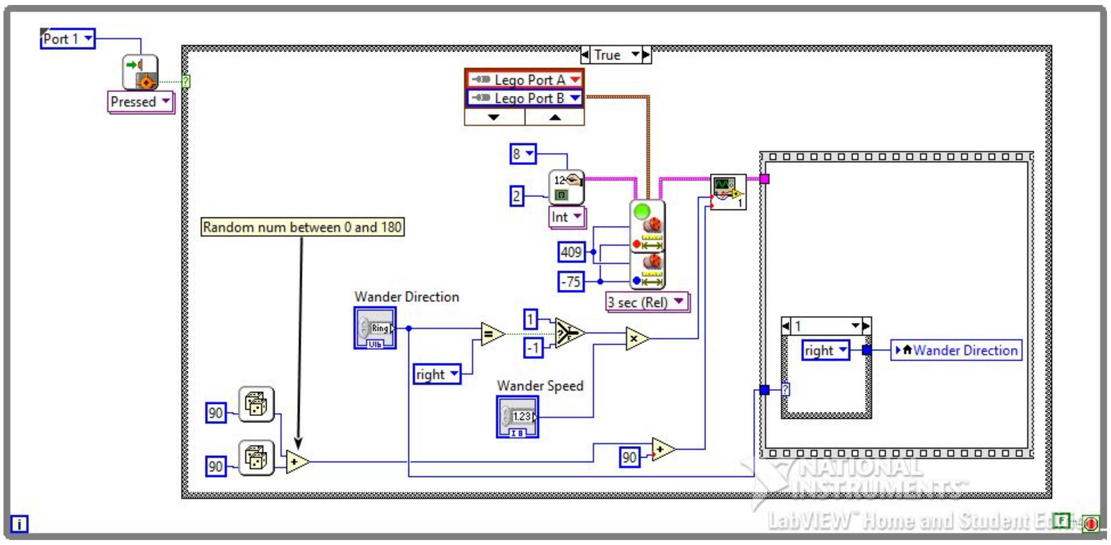
When this was done, the program was working as expected.

Behavior 2

This behavior also had a condition for entering, the condition was that the bumper detects an obstacle, which means that the bumper button has been pressed.

The actions that this behavior does are running backwards for 20 cm, then rotate in a random angle between 90º and 270º in the direction specified by “Wander Direction” and with the speed in “Wander Speed”. To finish, it changes the value in “Wander Direction”.

The block diagram is the following:



Here it can be seen that I used the flat sequence structure, and the reason for this is to avoid the strange situation where the Wander Direction is first changed and then is read for setting the speed. It should never occur that situation, but sometimes in real life problems, the impossible situations happen because some casualties.

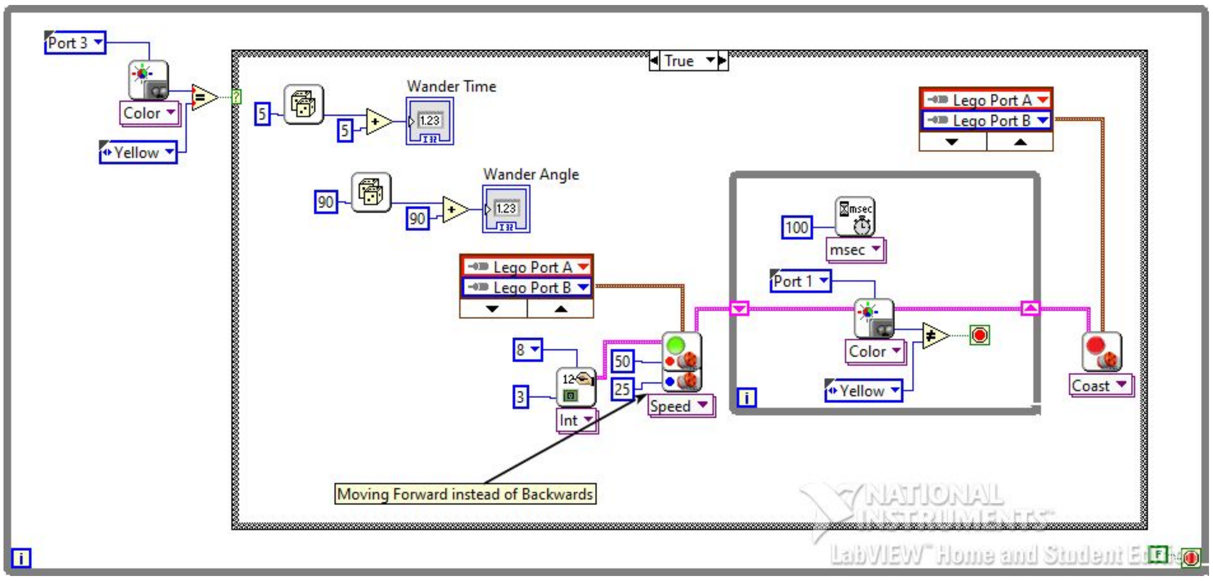
In order to make the robot reverse 20 cm, the fixed distance block was used. This block had one difficulty when using it, which is that the value of distance is not in any representation of meters, it is the encoder counts, so if the distance is set as 100, it will spin until the encoder reaches 100 counts, and the distance will depend on how are the wheels. The calculations for getting the value are the following:

- We know that at every spin there are 360 pulses.
- In each wheel spin the distance travelled is 2·pi·wheel radius.
- The radius of the wheels of the robots used are of 28 mm.
- So 360 pulses/spin divided by 2·pi·28 mm/spin equals to 2.05 pulses/mm.
- A total of 200 mm is wanted, so 2.05·200 equals to a total of 409,25 pulses to be counted, as pulses have to be an integer, the value of the fixed distance will be 409.

Like with the last behavior, in this I also had problems with the random number generator, and this time was due to a different factor. In the random value for the angle generation, as the angle is between 90º and 270º, there is needed a random number between 0 and 180 to be added to the 90º, but the random number generator can generate a number between 0 and 100 maximum, to solve this, I decided to use two random generators from 0 to 90 and add them, so they will generate a random number between 0 and 180 as desired.

Behavior 3

The triggering condition of this behavior is that the color detection find a yellow patch. When this occurs the actions that will be executed are moving forward in an arc until the yellow patch is left and then stop. The original activity was to move backwards in an arc, but if this is done, almost immediately after seeing the yellow patch it will drive almost nothing and then it will stop, doing this will only make sense if the sampling frequency of the color detector is very low, here is 100 ms, what is pretty big in our application. This behavior will also set the “Wander Time” into a value between 5 and 10 seconds, and will set the “Wander Angle” into a value between 90º and 180º.



To achieve the arc movement, I made the motors run at different speeds, as there was not any specification about how the arc should be or which direction I simply put two values with enough difference between them to appreciate the arc movement.

Alike in the behavior 2, here I did not use the sequence structure for setting the values of the Wander Speed and Wander Angle, the reason of this is that in this behavior these two values are not used, so changing their values earlier or later will not make any difference.

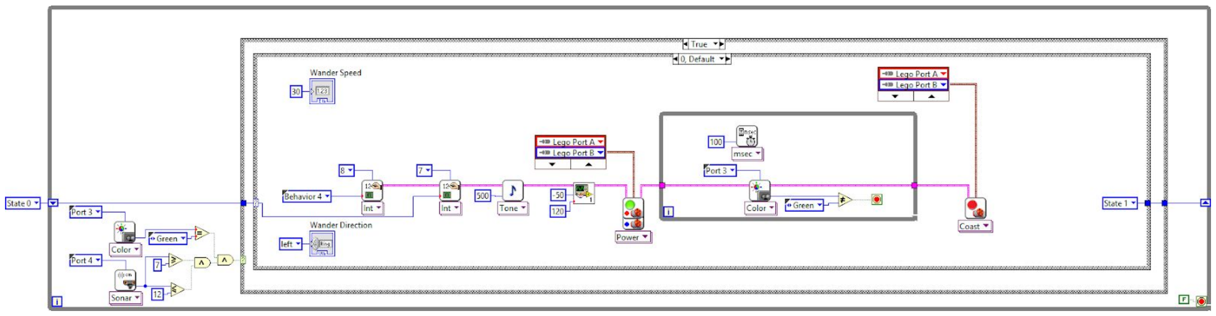
In this behavior a problem occurred, and it was that while checking the color in the color sensor, if there is no time delay, the motors do not run properly, so I had to add the 100 ms time delay in order to have the behavior working correctly.

Behavior 4

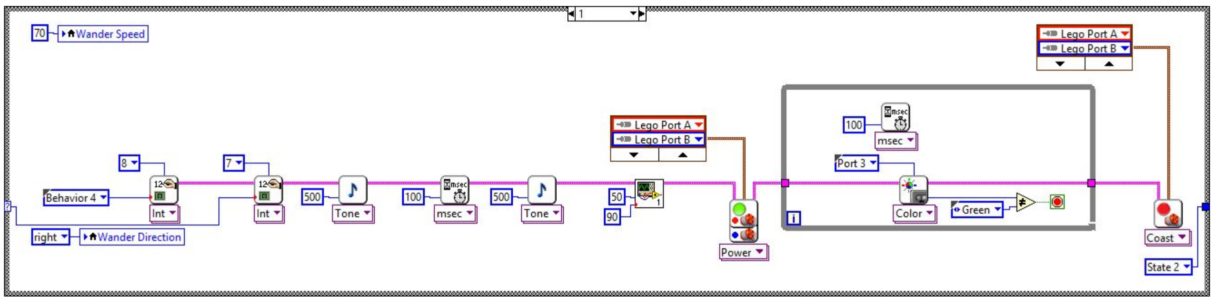
This behavior was the most complex one by far. It had a state machine with 4 states, each of them doing quite similar things, but with their own little differences. Every time that the triggering action was activated, a run of the state machine was done, which means that if the triggering situation is encountered more than one time, the actions will be different each time. This triggering action was that the color sensor senses a green patch and the distance sensor measures an object between 7 and 12 cm.

In all the states apart from printing the behavior number, as usual in all the behaviors, they also print the actual state of the state machine. Below, there is an explanation of the actions of each state.

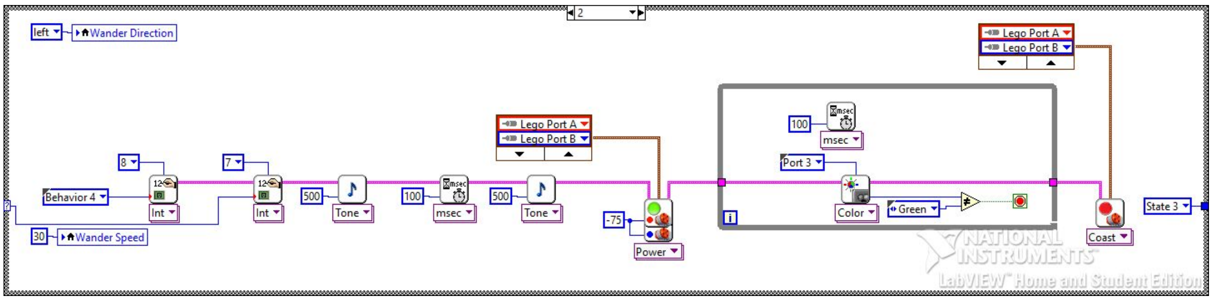
The state zero, generates a tone of 0.5 seconds, then it makes the robot rotate 120º to the right, and makes it move forward until the green patch is exited, it also changes the Wander Speed to 30 and the Wander Direction to left, and changes the state to 1. The block diagram can be seen below:



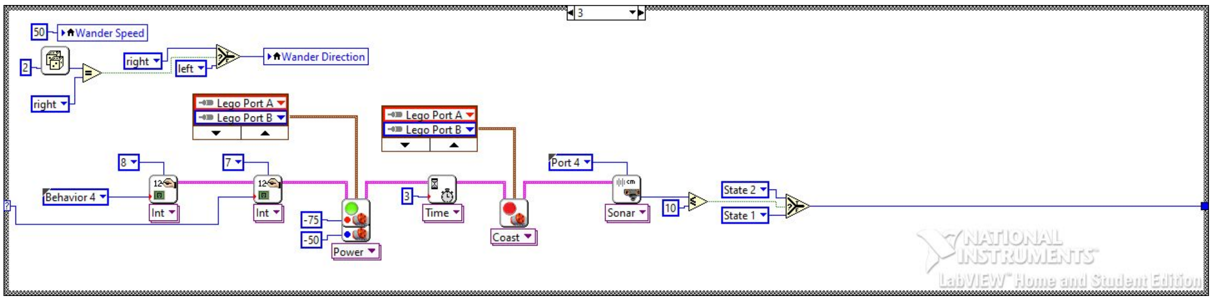
Then, the state 1, generates two tones of 0.5 seconds each, then it rotates 90º to the left, and it runs forward until the green patch is exited, also the Wander Speed is changed to 70, and Wander Direction to right, and it changes to state 2. The block diagram is the following:



The state 2, it starts just as the state 1, but this one after the tones, it drives the robot backwards until the green patch is exited, also it changes the Wander Speed to 30, and Wander Direction to left, and then it changes the state to 3. Below there is the block diagram of this state:



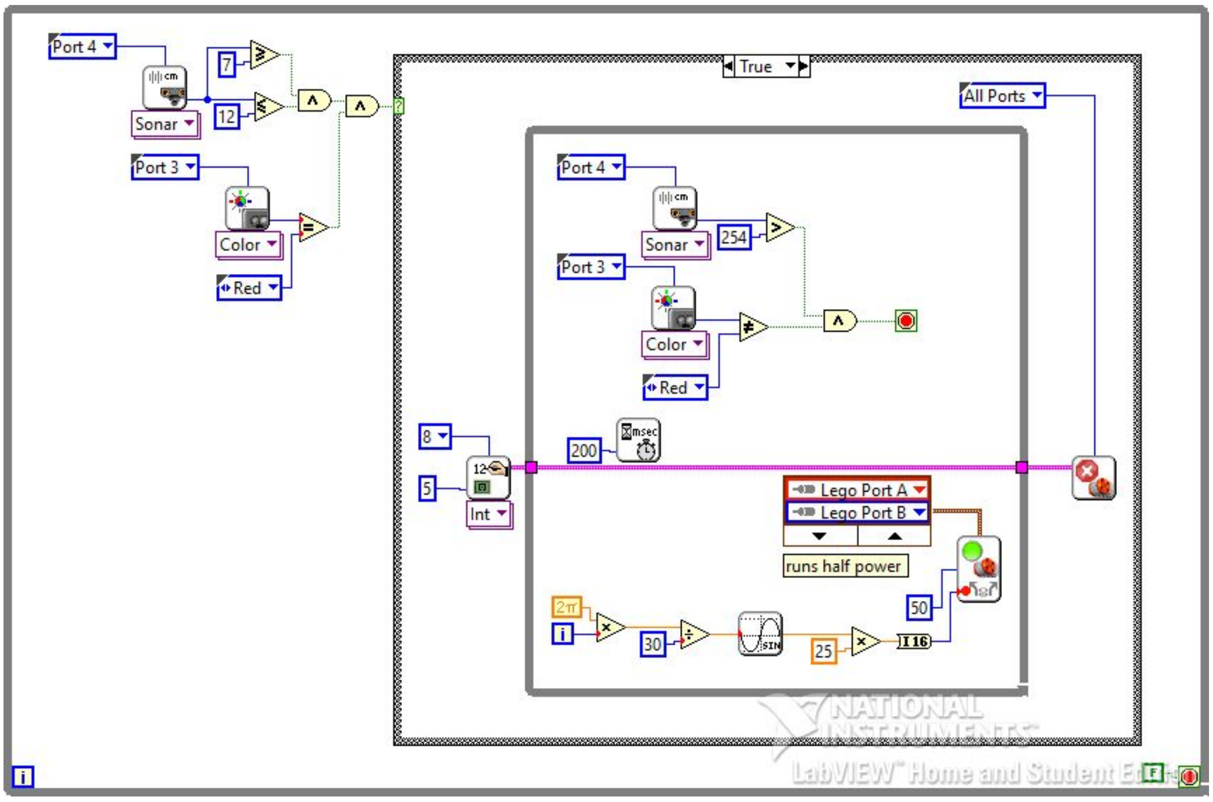
The last state, basically moves the robot backwards in an arc for 3 seconds, after this, if the distance sensor detects an object less than 10 cm, it changes the state to 2, and if not, it changes the state to 1. It also changes the Wander Speed to 50, and the Wander Direction to a random direction. The block diagram for this state is the following:



This behavior had difficulty due to the multiple things that it had to do, and take into account in each state of the state machine, but it did not cause any problem, as it did not have a lot of states and also it did not have sub states, which would enlarge the complexity a lot.

Behavior 5

This behavior was entered when the sonar detects an object between 7 and 12 cm, and the color sensor detects a red patch, it did not have any special activity, there was only needed to redo the "snake" behavior supplied by the NXT library, to finish when the red patch is left and the sonar does not detect any object, which means a reading of 255 in the sonar. The block diagram is the following:



When the stopping condition was set, and I tried its working, surprisingly it did not work, this was due to the fact that the steering block by default in the "snake" behavior, was not the correct for moving the robot, so I need to change it to the steering block. Also, I changed the input and output of the sine signal, in order to get a movement with a specific frequency and amplitude. In this case, every 30 loops it will do a complete wave, as every loop lasts 200 ms, every 6 seconds does a complete sine period.

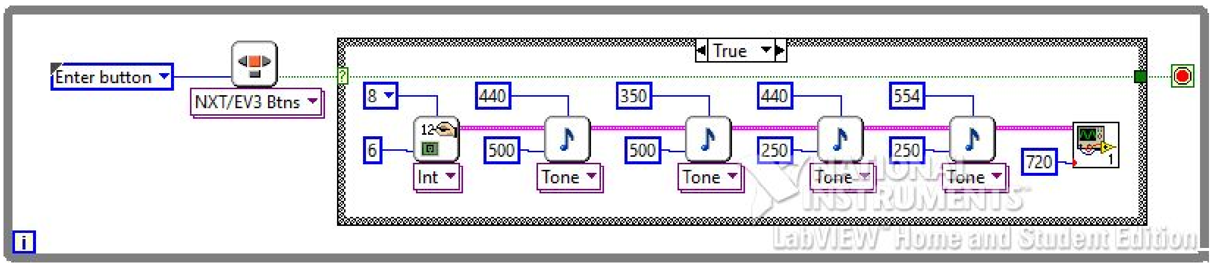
Behavior 6

This is the termination behavior, like the initialization it will only be executed once, but the difference between them is that this one will do some actions in order to let the user know that the program ended.

Its triggering conditions will be pressing the orange button from the robot, and as this behavior has the highest priority, a touch of that button will always mean executing this behavior and a termination of the program.

These actions that it will do are playing a victory tune, which I decided to play 4 notes to create a very simple melody, but by hearing it can be acknowledged that the program has finished, and then spin the robot twice, which means a total rotation of 720º.

The block diagram is the following:



As this behavior does not have any special or complicated actions to do it did not create any problem at all.

Framework

After having all the behaviors working as different VIs, it was time to start working in the framework, as it was explained before, to join all the behaviors into one program, we used subsumption architecture, this needs different layers with priorities, which will consist of the different behaviors, and then it needs an arbiter in order to deal with the control requests of each behavior to decide to give or not the control to the behavior requesting it. To implement that, there were some examples provided, I used one of those examples as a template to work with.

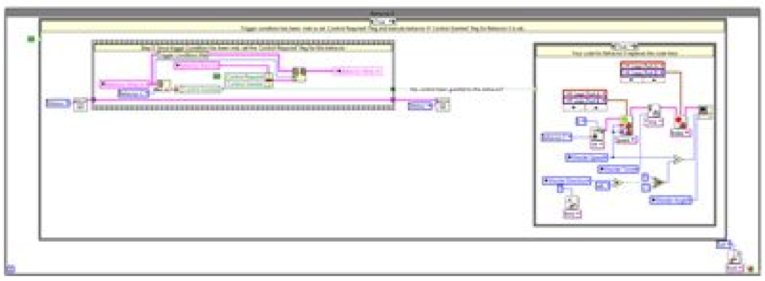
The examples had everything implemented, the initialization and termination parts, and in the running part they had the termination routine, which is the only way to finish the program, and also they had the arbitration system implemented. The work that was needed to be done was basically inserting the block diagrams of each behavior in the correct part, and set the desired triggering conditions for each behavior, and get everything together.

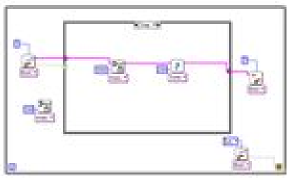
For the behaviors from 0 to 3 and 5, it was the same procedure for all of them, copy the block that had the arbitration, and then inside, copying the block diagram done before and in the triggering part copying the trigger also seen before. But, for the behavior 4 it was not that simple, as this was a state machine, in order to make it work, I had to do the state passing with shift registers, which was not much of a problem. The behavior 6, as an exception, instead of having its block like the others, I copied its code in the termination part, as this behavior is the termination one, and it will be useless to do all the processing for this when it is known that it will only execute once.

In order to build the program correctly, I started with only the behavior 0, and when it was working I implemented the behavior 1, and so on. Until I reached the behavior 5 working, then I inserted the blocks of behavior 6 in the terminating part. This is the way how the subsumption architectures should be built.

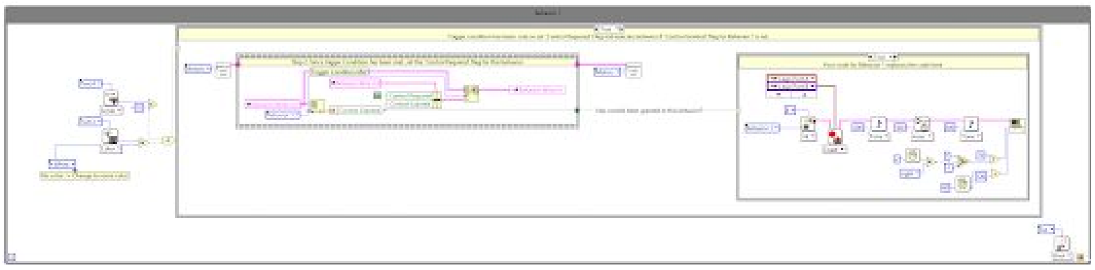
Each behavior block code which was inserted in the program can be seen in the following list:

- **Behavior 0:**

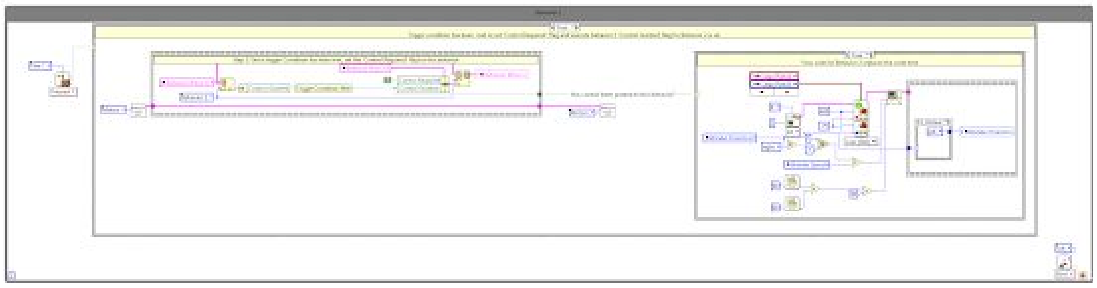




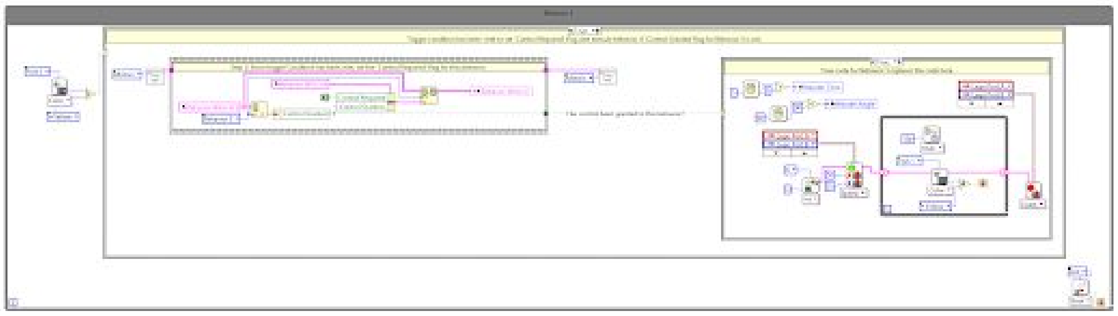
- **Behavior 1:**



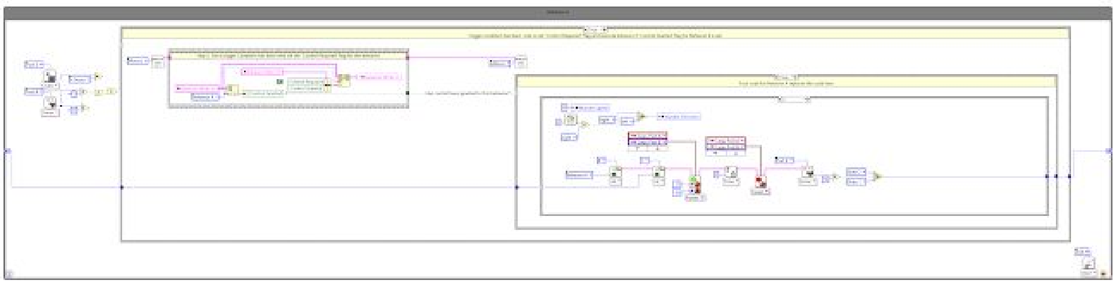
- **Behavior 2:**



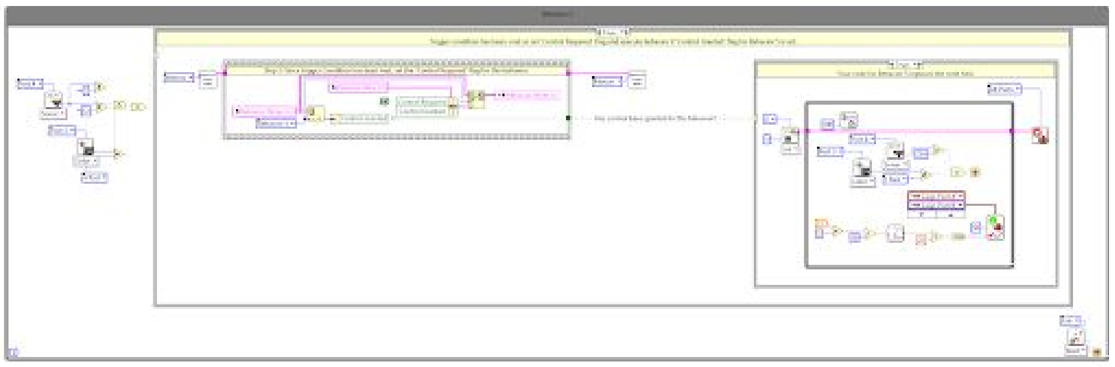
- **Behavior 3:**



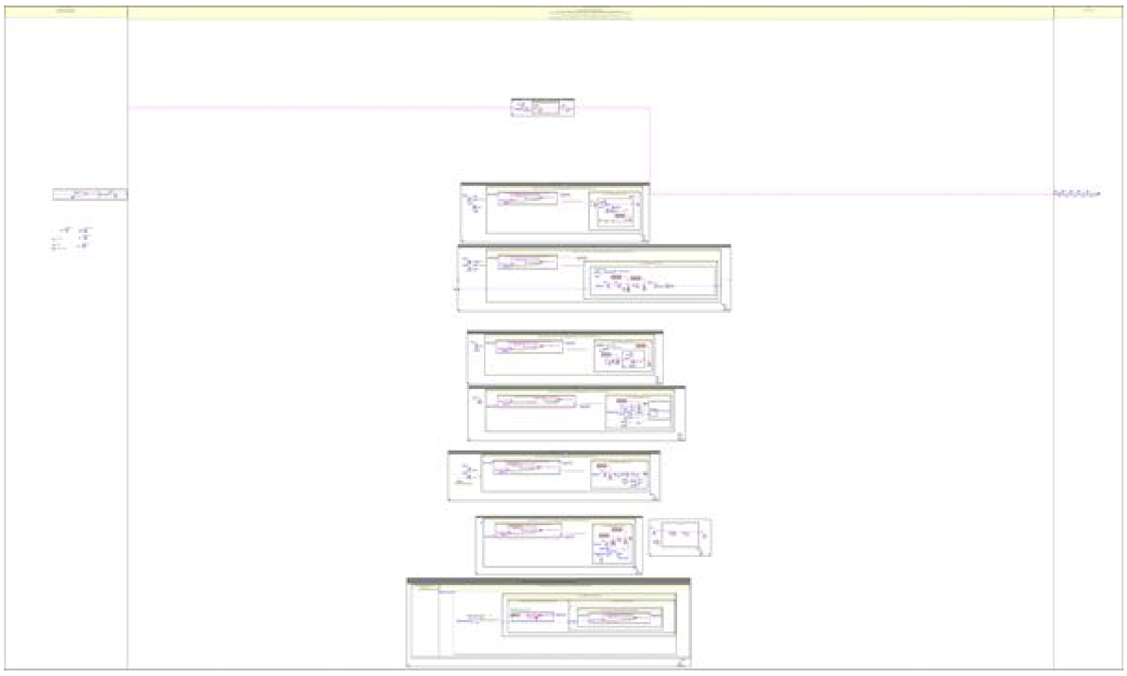
- **Behavior 4:**



- **Behavior 5:**



- **General:**



I have not inserted the image from behavior 6, as it is separated into two parts, the stopping program part, and executing actions part. Both parts can be seen in the image above.

After everything was implemented and working, it only lasted to see if the robot acted as a clever unit or not. When I deployed the program in the robot, and left it by itself, it was seen that it was working almost as expected.

The only unexpected occurrence was that when pressing the orange button, the program was not terminating immediately, it needed to finish the execution of behavior 0 just before exiting. This happens because who handles this is not the arbiter, it is a flag that terminates the behaviors.

Testing and results

In order to test the project I decided to go step by step. After having each behavior done as a separate VI, I tested if it was working as expected or not in some different runs of the program in order to see if any unexpected situation occurred. For the behaviors, what was needed to be tested was if they were doing the actions or not, and if the triggering was working correctly. Then until the program was working as expected, I did not continue doing the next one. After all the behaviors were doing what they were expected to do, then I started with the subsumption architecture.

The subsumption, as I explained before, I built a new behavior only when the lower priority one was finished, which means that every time a behavior was introduced, a test of its working was done. The tests were exactly the same as for the behaviors alone. Until it was working the newer ones were not implemented, as the behaviors were tested as separated VIs, and they were working correctly, not many problems occurred, so everything was working correctly at almost the first time.

All the tests carried at the last part were successful, so the result of the program was correct, in some middle tests, the results were wrong, what meant to do some changes in the implementation of the behavior.

A fact that must be taken into account is that all the tests and building of the behaviors were done with a set of connections with the ports of the LEGO NXT and the sensors, if these connections were changed, the behaviors will not work as expected.

Conclusions

In the real world, this project will have started with the desire of making the robot run in a room, and do something special, which would have been a new behavior on the top of the behavior 5, and the behaviors implemented would be responsible for avoiding obstacles and wandering around, and the new behaviors will do other tasks of utility.

This project helped to see how the subsumption architecture works, as the final part is mounting one single program with this architecture. Also, it helped to see the process of working, in this case the initial steps were given by the project description, as in the real world, first we would have had an initial idea of how the robot should work, then this idea should be divided into behaviors doing different actions, with their own triggering actions and different priorities, what it was provided. Then, in the project we had to work in the following steps, which are designing each behavior alone, and then when all work correctly, the behaviors are joined together in the architecture.

When finishing the whole program, is good to see how a group of different programs that seem to not have any relation, are running the robot cleverly.